Esercitazione API 31/5/22

es.: Minimo Antenato Comune (MCA)

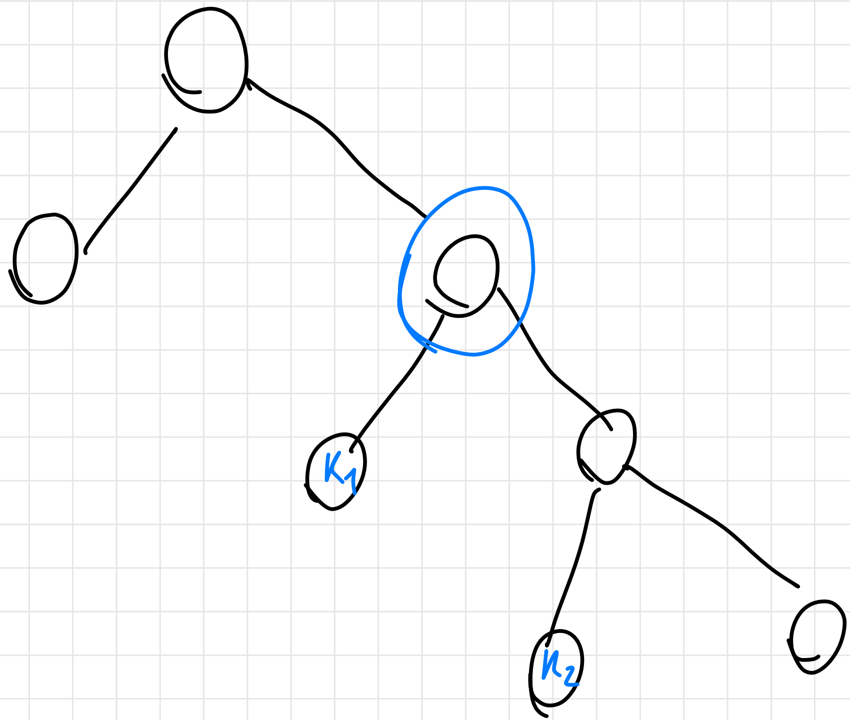
Input: due chiavi k_1, k_2
un albero binario

Output: MCA dei nodi con chiave k_1 e k_2

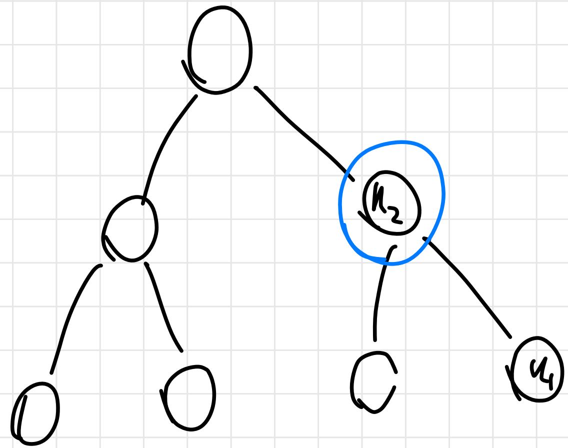
Assunzioni:

- Albero non ha chiavi duplicate
- k_1, k_2 sono presenti nell'albero

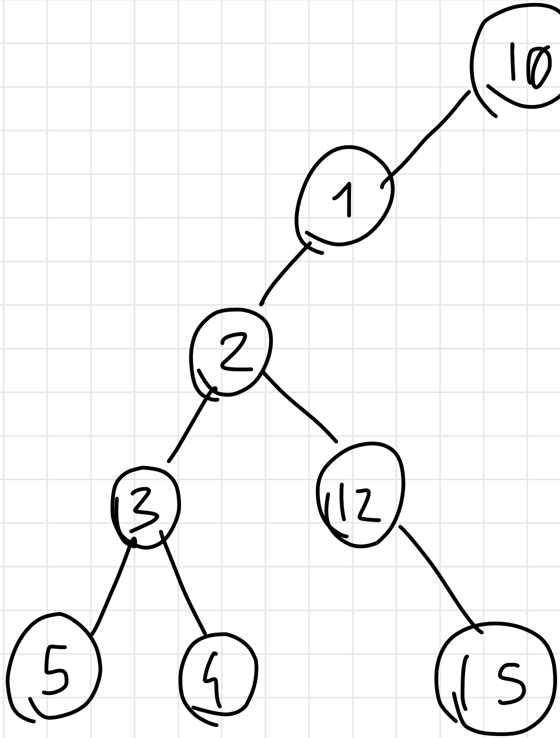
Def.: MCA è un antenato comune ad entrambi i nodi e più lontano possibile dalla radice



K_1
K_2



n_2
n_4



$k_1 = 5$

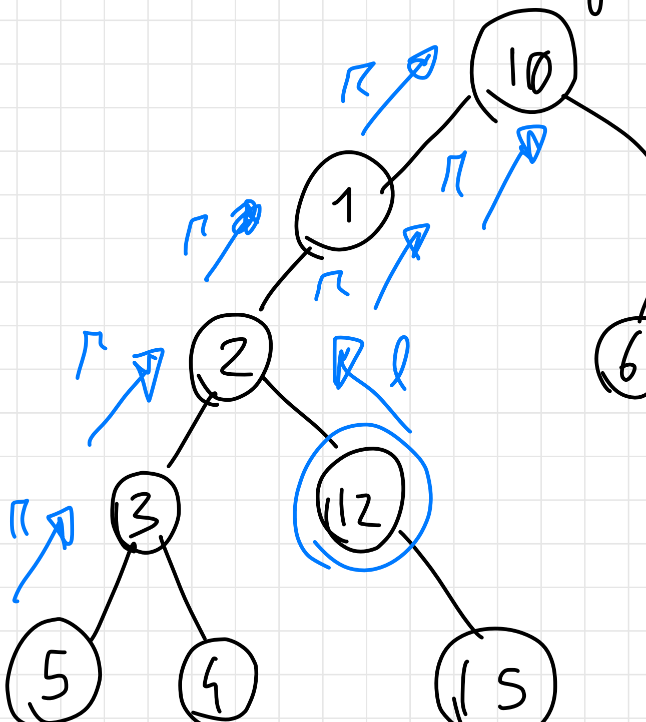
$k_2 = 12$

Trovo nodo k_1

Salgo fino alla radice

Salvo una stringa : 'l' se salgo verso sx

'r' verso dx



5 : r r r r

12 : l r r

$k_1 = 5$

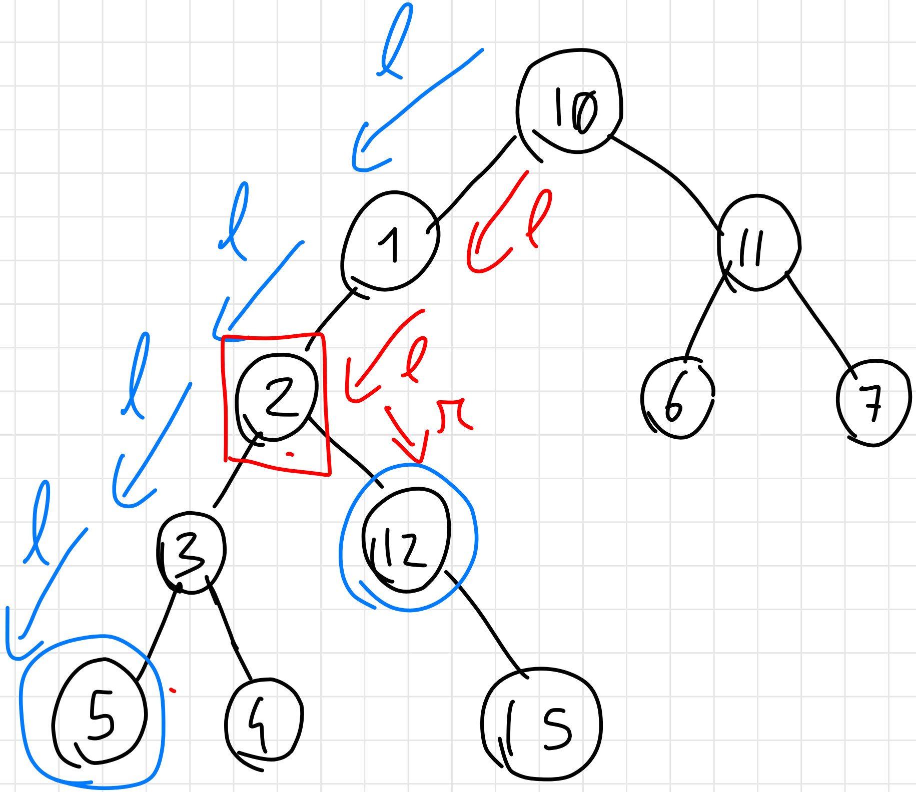
$k_2 = 12$

A.p , A.left , A.right

Seconda idea:

modificare la ricerca in maniera che si tenga in mem. il percorso che ci porta a k_1 (modifica algo. InOrderVisit)

Stessa cosa per la ricerca di k_2



l l | l l

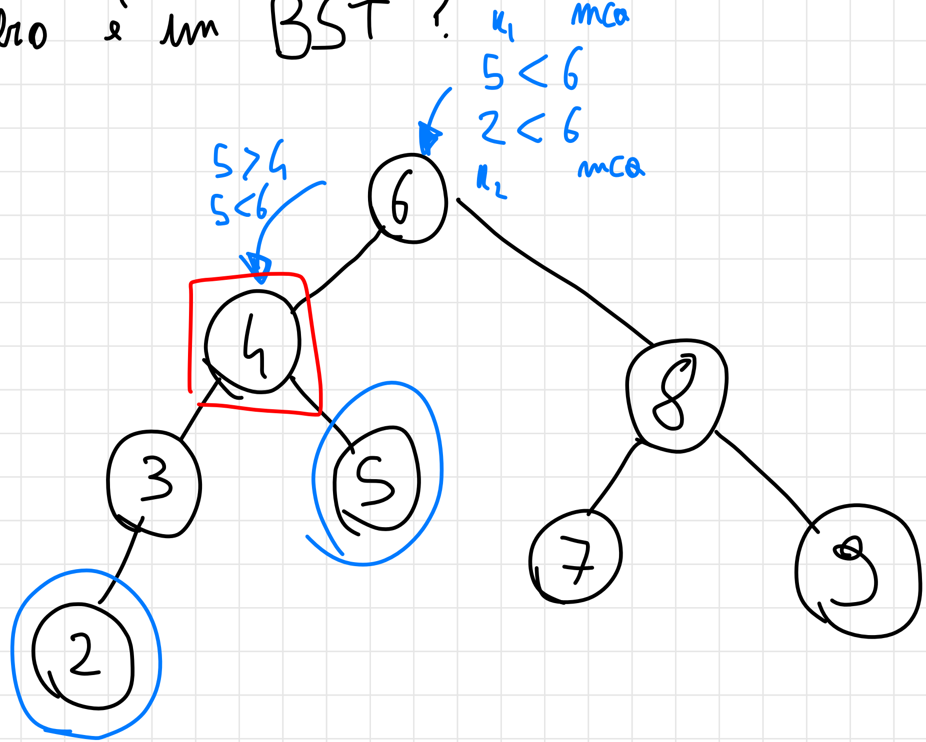
l l | r

$k_1 = 5$

$k_2 = 12$

$T(n) = \Theta(n)$

Se l'albero è un BST?



```
MCA(T, k_1, k_2):
    mca ← T.root
    while mca ≠ NIL:
        if mca > k_1 and mca > k_2 :
            mca ← mca.left
        elseif mca < k_1 and mca < k_2 :
            mca ← mca.right
        else
            return mca
```

$T(n) = O(h)$

Se BST bilanciato:

$T(n) = O(\log(n))$

Se sbilanciato

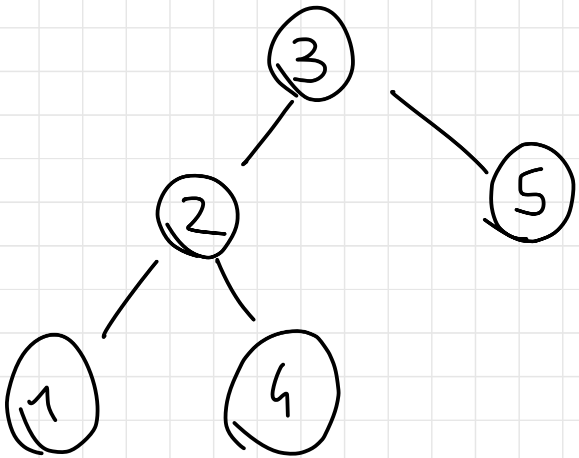
$T(n) = O(n)$

Es. Dato un albero binario, controllare se è un BST

1ª idea: isBST(T): $\qquad T(n) = \Omega(n)$

```
if T = NIL:
    return True
if T.left ≠ NIL and T.left.key > T.key:
    return False
if T.right ≠ NIL and T.right.key < T.key:
    return False
if not isBST(T.left) or not isBST(T.right):
    return False
return True
```

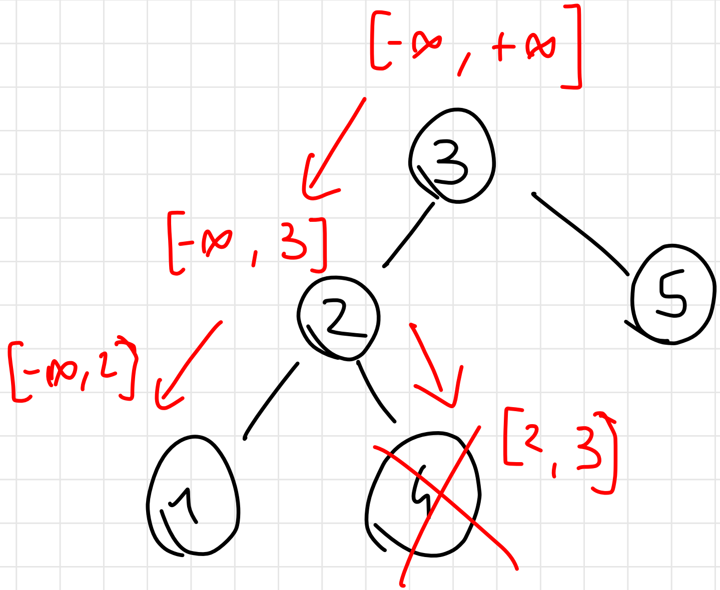
Controes.:



Il nostro algo. ritorna True, ma non è BST

```
isBST(T):
    if T = NIL:
        return True
    if T.left ≠ NIL and max(T.left) > T.key:
            return False
    if T.right ≠ NIL and min(T.right) < T.key:
            return False
    return isBST(T.left) and isBST(T.right)
```

$$T(n) = n \cdot T_{min/max}(n) = n \cdot n = \Theta(n^2)$$

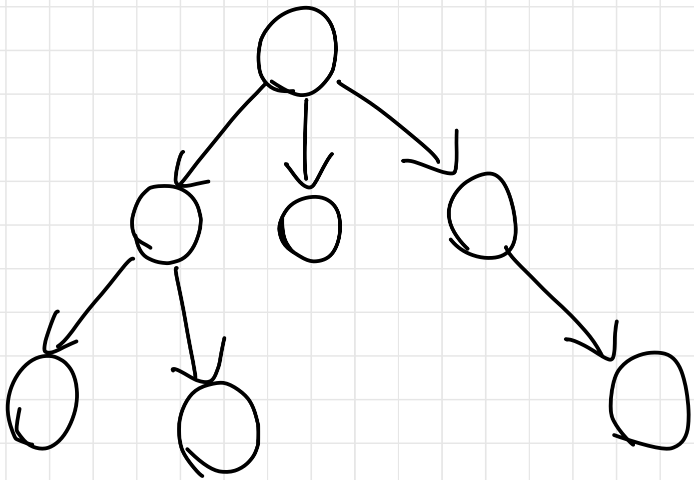

[-∞, +∞]
3
5
[-∞, 3]
2
[-∞, 2]
1
4
[2, 3]

```
isBST(T, min, max)                              T(n) = Θ(n)
    if T = NIL:
        return True
    if T.key < min or T.key > max:
        return False
    return isBST(T.left, min, T.key) and
           isBST(T.right, T.key, max)
```

Sol. alt. : InOrder visit, salvando in una lista i nodi visitati + controllo se la lista è ordinata

$$T(n) = \Theta(n)$$

Es. : Progettazione Albero non binario di ricerca



Ogni nodo ha questa struttura :

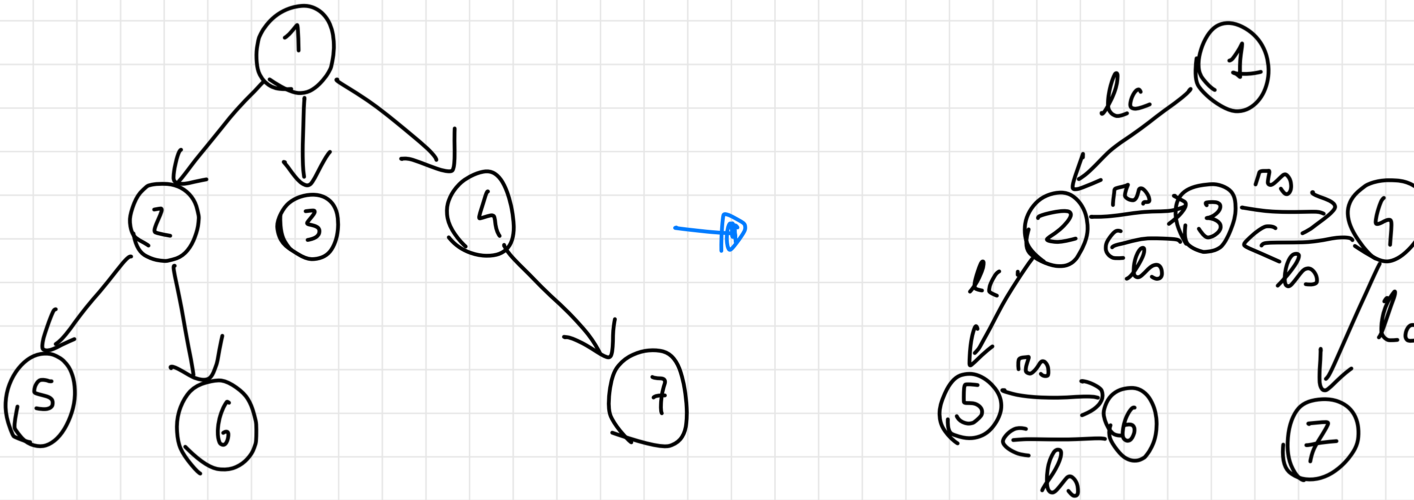
n. key

n. lc : punt. figlio più a sx

n. rs : punt. primo fratello a dx

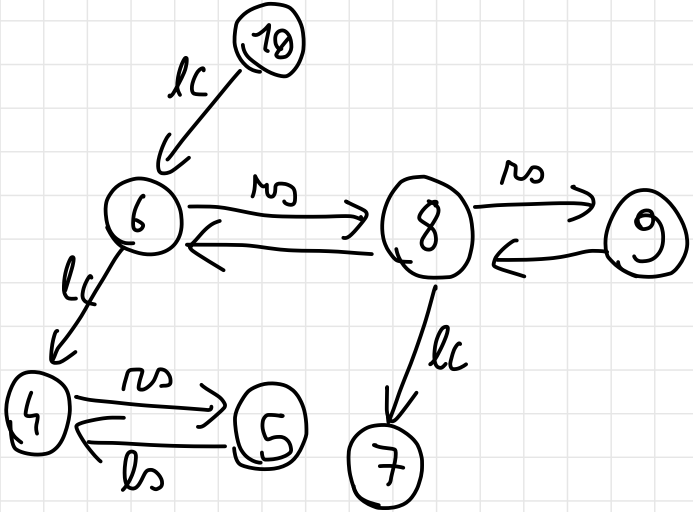
n. ls : punt. primo frat. a sx

n. p : padre



Richiesta: organizzare le chiavi nell'albero in una maniera tale da avere un criterio per la ricerca (come nei BST)

Implementare: Search, Successor, Pred., Insert, Delete



Organizzaz. chiavi:

tutte le chiavi del sottoalb. con radice in T.lc. sono < T.key

tutte le chiavi del sottoalb. con radice T.rs sono > T.key

```
Search(T, k):
    node ← T.root
    while node ≠ NIL:
        if node.key = k
            return node
        if k < node.key:
```

node ← node.lc
else
node ← node.rs

return NIL

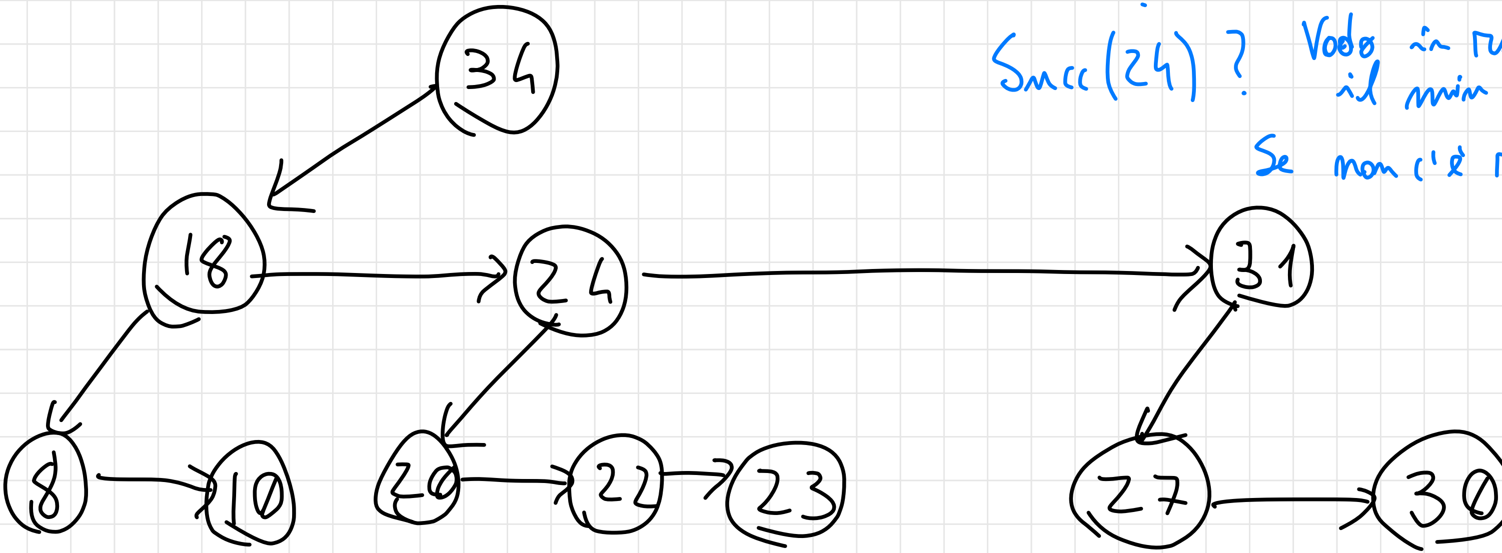
search(20) ✓

min? il figlio più a sx

max? root

Succ(24)? Vado in rs e trovo il min

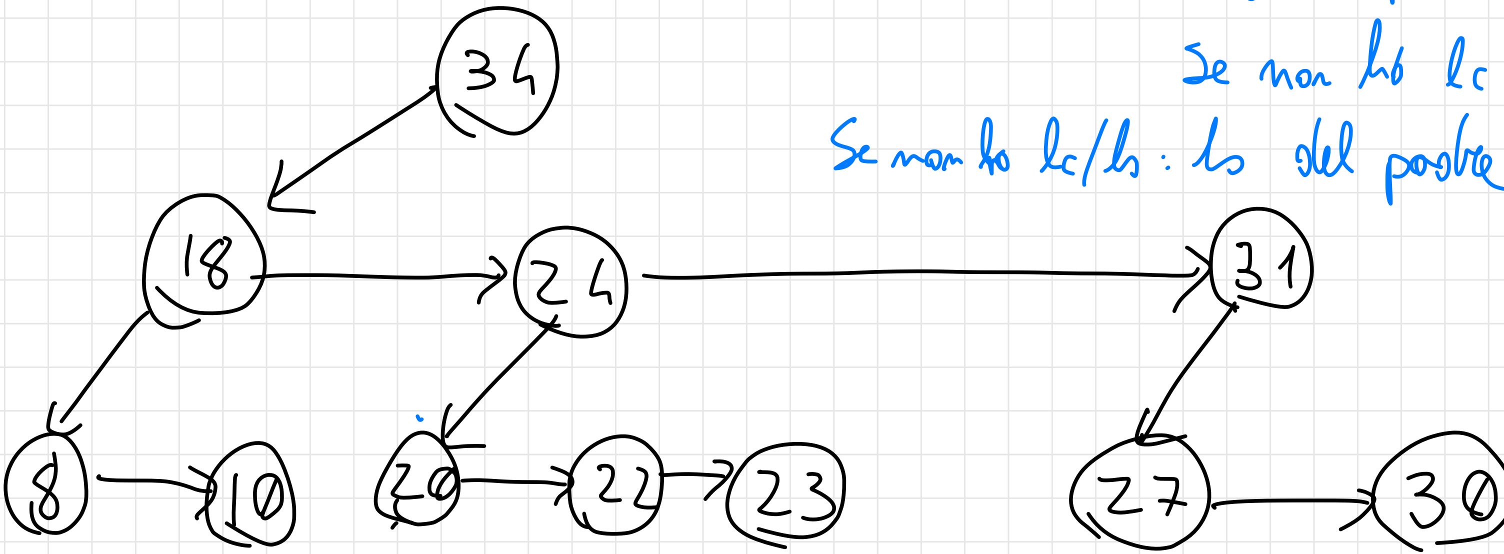
Se non c'è rs, padre



Pred(24)? Prendo lc e trovo il max (fratello più a dx)

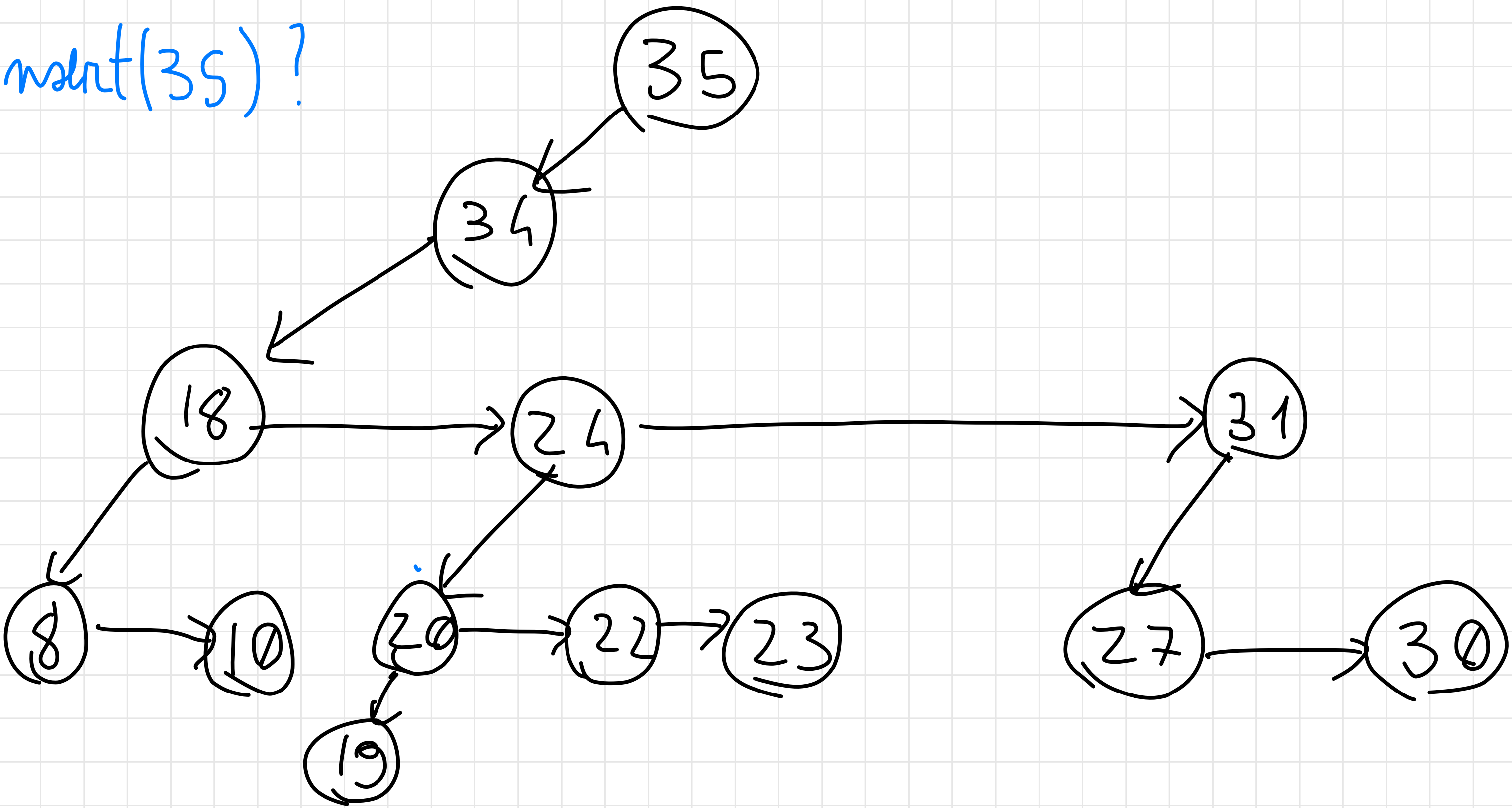
Se non ho lc, ls

Se non ho lc/ls: ls del padre



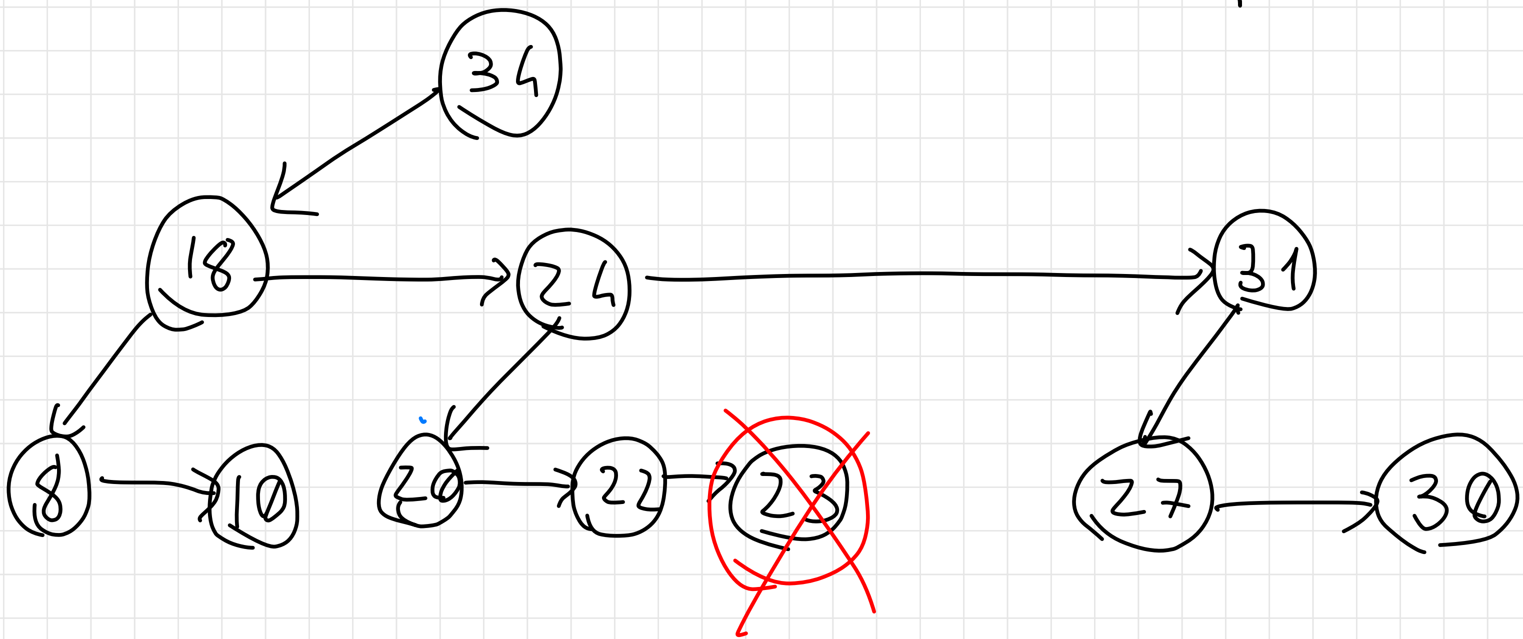
Insert(19)? Search 19, inserisco dove dovrebbe essere

Insert(35)?



Delete: Caso 1: nodo senza figli né fratelli rs

→ elimino senza problemi

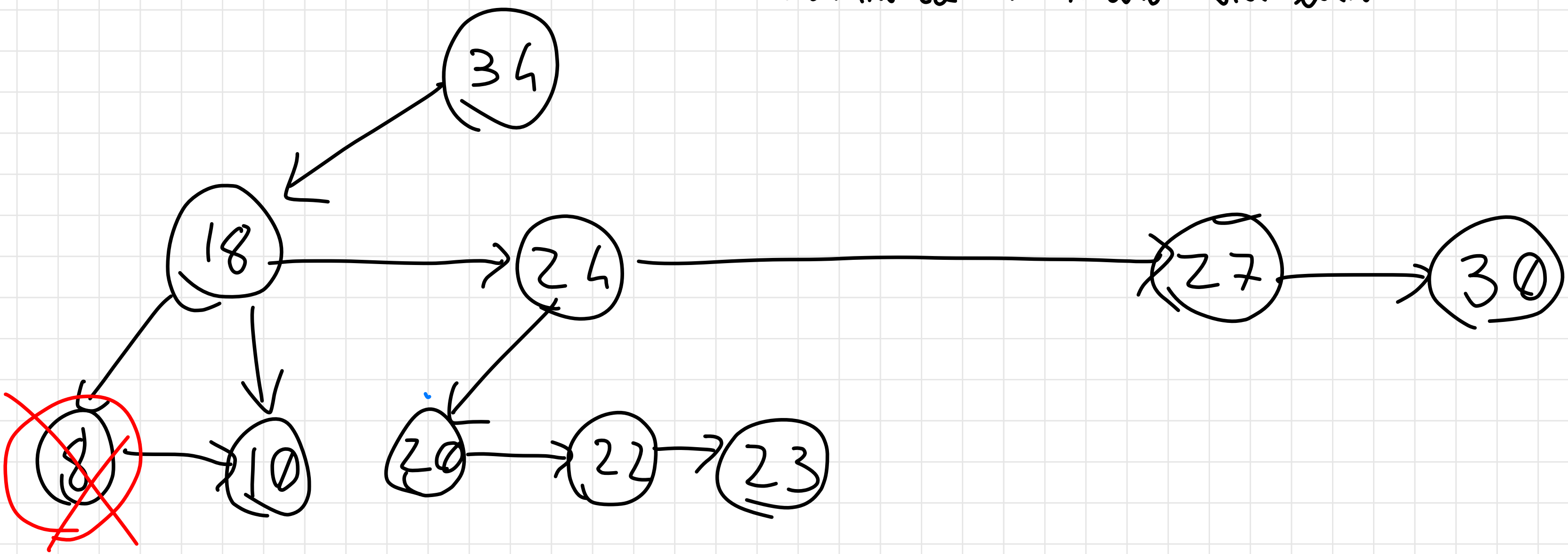


Delete: Caso 2: 2.1 nodo con lc, no rs

lc sostituisce il nodo da eliminare

2.2 nodo con rs, no lc:

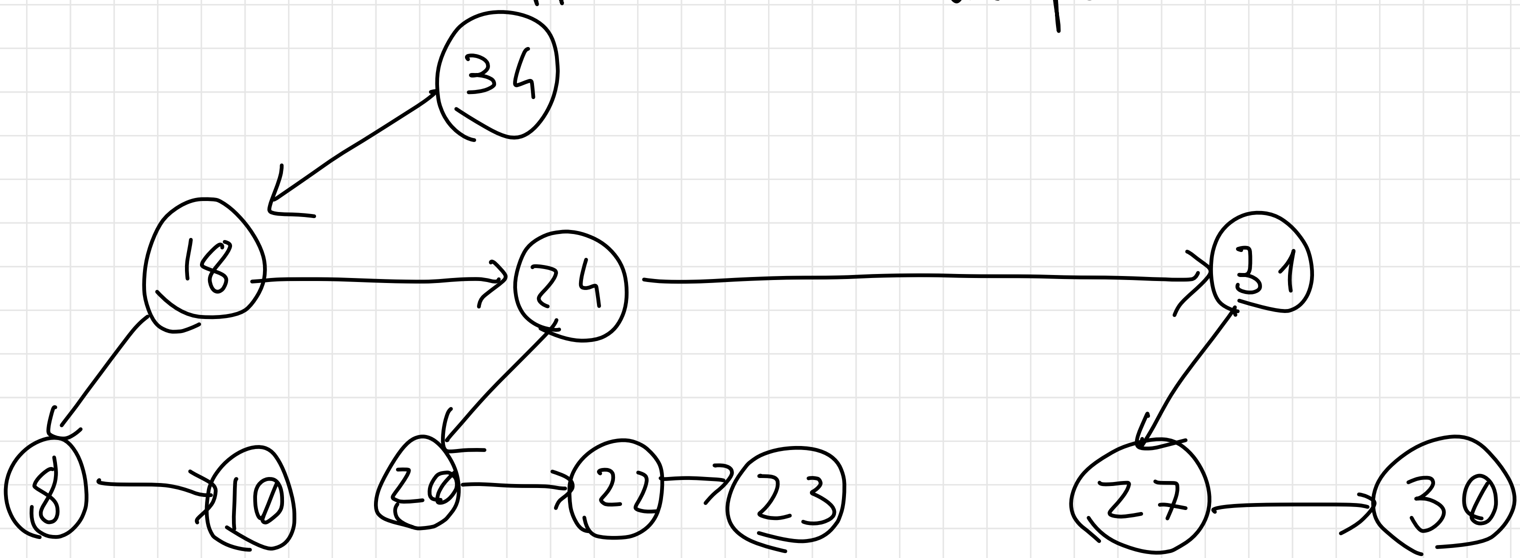
rs sostituisce il nodo da eliminare



Caso 3: Nodo ha sia rs che lc

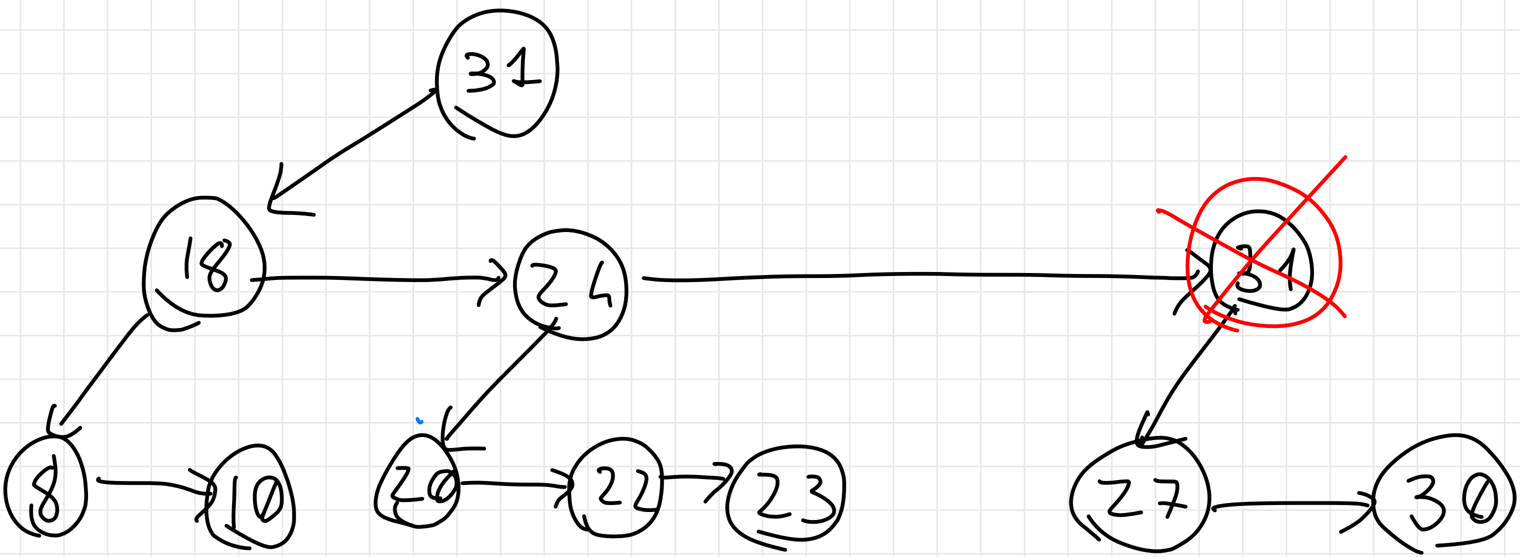
Cerco pred., lo copio nel nodo corrente

applico la delete al pred.



Caso 4: Elimino Radice

Delete: Come Caso 3



Intro Dizionari

Insieme dinamico (key, value)

Per accedere ad un dizionario si usa la key

Search (T, k) , Insert , Delete

U : Insieme di tutte le possibili chiavi

$|U|$ può essere molto grande

Prima implementaz. : Tab. a ind. diretto (Array)

Array lungo $|U|$

insert(T, k, v):

$T[k] \leftarrow v$

search(T, k):

return $T[k]$

$U = \{1, 2, \dots, 10\}$

$$T: \begin{array}{c} \begin{matrix} 1 & 2 & \dots & 10 \end{matrix} \\ \left[\begin{matrix} NIL, & NIL, & \dots, & NIL \end{matrix} \right] \end{array}$$

$|U|$ molto grande

Num. di chiavi da ins. : m

$$\boxed{m << |U|}$$

Funz. hash:

$$\boxed{h : U \longrightarrow \{1, \dots, M\}}$$

Creo una tabella grande M. $M << |U|$

insert(T, k, v):

$T[h(k)] \leftarrow v$

search(T, k):

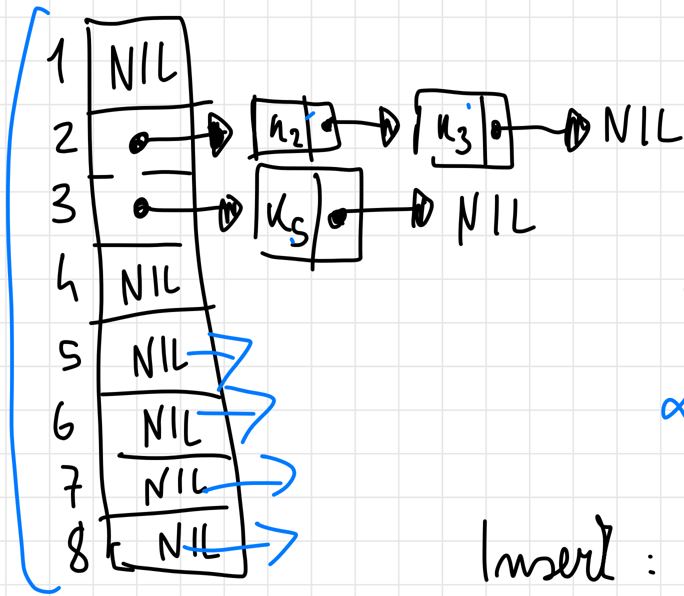
return $T[h(k)]$

$M << |U| \Longrightarrow$ ci sono collisioni

$\exists\, k_1 \neq k_2 :$

$h(k_1) = h(k_2)$

Risoluz. collisioni: Chaining



k_2 $h(k_2) = 2$

k_3 $h(k_3) = 2$

k_5 $h(k_5) = 3$

$\alpha = \frac{3}{8}$

$\alpha = 1$

Insert : $\Theta(1)$ (con puntatore alla coda)

Search : $\Theta(|T[h(k)]|)$

M dim. tabella

n elem. memorizzati

$$\alpha = \frac{n}{M}$$ fattore di carico

Search: $\Theta(1+\alpha)$

In pratica: $M = \Theta(n) \Rightarrow \alpha = \Theta(1)$

Search $= \Theta(1)$

Risol. collisione: Ind. Aperto

1	NIL
2	~~k_1~~
3	k_2
4	NIL
5	k_3
6	NIL
7	NIL

$k_1 : h(k_1) = 2$

$h(k_2) = 2$

$h(k_3) = 5$

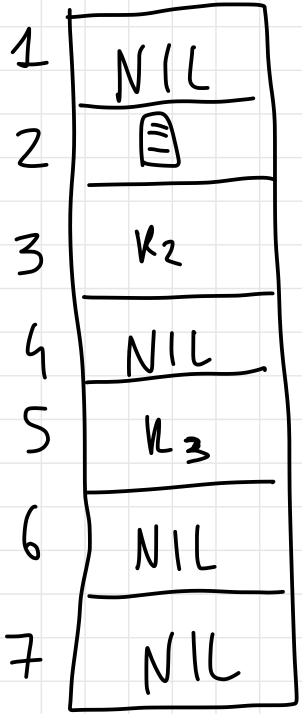
$h'(k, i) = h(k) + i$

ispezione

$$\alpha = \frac{n}{M} \qquad n \cancel{>} M \qquad \alpha \leq 1$$

search: $\Theta\left(\frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right)\right)$ se $\alpha < 1$

$M = 2n \Rightarrow \alpha = 1/2$ $\qquad \Theta(1)$



search k_2 ✓ $h(k_2)=2$

delete k_1

→ search k_2

TdE

Input: Array A di n interi distinti

Valore intero X

Out: $\exists$ due elementi i, j t.c.

$$A[i] + A[j] = X$$

Progettare un algoritmo e una struttura dati che min. $T(n)$ nel caso medio, senza eccessivo spreco di mem.

nell'array A ci sono val $\in$ [MININT, ..., MAXINT]

No tabella grande $|MAXINT - MININT|$

Creo tabella di hash

for $i: 1 \to n$:

insert$(T, A[i])$ $\Big] \; \Theta(n)$

Per $i: 1 \to n$:

ricerco $X - A[i]$ in T $\Big] \; n \cdot T_{search}$

Scelgo io T

$\boxed{\begin{array}{l} T \text{ chaining} \\ |T| = n \end{array}}$ $\to \Theta(1+\alpha)$, $\alpha = \dfrac{n}{|T|} = 1$

$\to \Theta(1)$

$$T(n) = \underbrace{\Theta(n)}_{\text{crea tab.}} + n\,\Theta(1+\alpha) = \Theta(n) + \Theta(n) = \Theta(n)$$